PIGRA

A Tool for Pixelated Graph Representations

Thomas Bläsius, Fabian Klute, Benjamin Niedermann

Contents

1 Introduction

PIGRA is a GUI-based tool for creating grid-based representations of graphs, using integer linear programming (ILP). In a grid-based representation each vertex and each edge of a graph can be described by a set of grid cells. The GUI allows you to easily combine pre-defined ILP-constraints in order to model grid-based graph representations. If the set of pre-defined constraints is not sufficient for your use case you can define your own constraints by using PGL, a small scripting language build into PIGRA. If you want to know more about the theoretical background please read [].

The following chapters should give you a fair overview over what you can do with PIGRA. In Section 2 we will dive into using the GUI and PGL following a first example. Section **??** starts with a more complex problem, namely finding visibility and k-visibility representations for a given graph. Finally Section 4 leaves you with a reference or cheat sheet for PGL.

2 A small example

The first example we will look at introduces some of PIGRAs most common features. Our goal is to load a graph and create a drawing, where every node is represented by a block of size four. Lets start PIGRA and load the graph first. For this go to File → Load Graph and open the file example1.gml. On the left side you see three tabs. Choose the graph tab, it should be at the bottom. Now in the center you should see a preview of the graph. You can see the graph consists out of six nodes and no edges. Going back to the first tab we want to start creating the PGL program needed to draw the boxes. In the center you find a box with Macros, open 2D and mark the checkboxes at check 2D Border, Begin End Columns and Begin End Rows. When selecting the first macro make sure to change its value for n to one on the right side. A Macro is a small program written in PGL and we will use them to get around some not really interesting technical problems. For more informations please take a look at []. Just to give an intuition, the first macro creates a border of size one around the grid and the other two set begin and end variables, which are needed to make sure nodes and edges are connected. One thing you will perhaps notice is that upon selecting the 2D Border macro another one was chosen too. This one you find under Default Variables and it defines the basic variables, which we will use too in a second. The next thing we want to do is set the grid size. This can be done under Grid Settings. A size of 7×7 should be enough to start with.

So far we did not need to write a line of PGL code, in fact we could create the boxes completely out of predefined macros, but for the sake of this example we will write the final bits by hand. On the write you find an editor. If we want boxes the first thing we have to guarantee is that every node has at least a four pixel representation. Now it is a good time to read the help for the the Default Variables macro, since we will use the x variable. To assure every node to be represented by four pixels we only need to write two real lines of code:

```
for $n$ in $nodes$ do
    sum $x(\_, \_, n) = 4$
end
```

Lets look at this little snippet. The first thing we do is iterate over all nodes in the graph. In the next line we build the sum over x restricted to the node n. The underscores mean, that the respective values are not. The easiest way to understand this might be to see the equivalent mathematical representation:

$$\forall n \in nodes : \sum_{r \in rows, c \in cols} x(r, c, n)$$

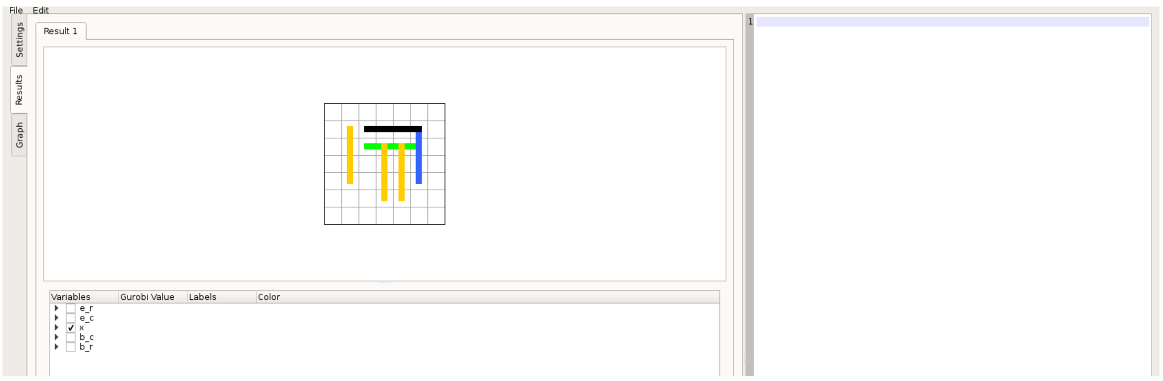
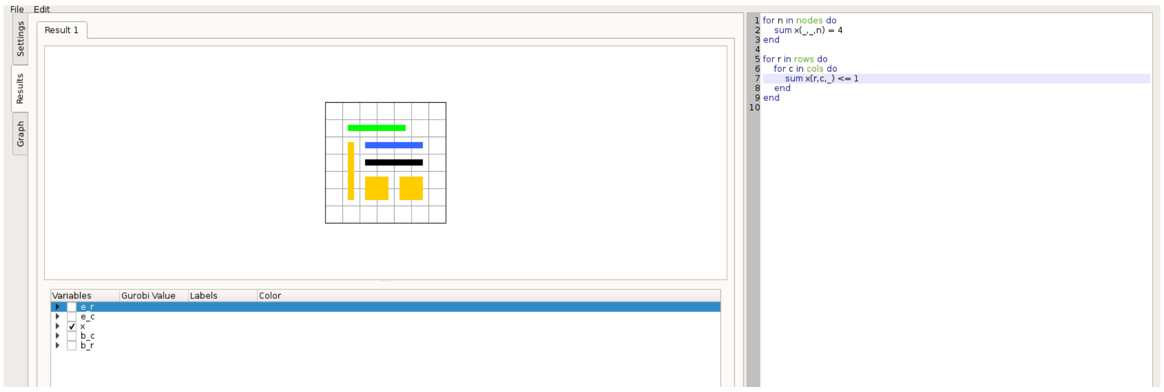


Figure 1: The result with overlapping boxes, still all nodes have only size four.



Variables	Gurobi Value	Labels	Color
☐ e_r			
☐ e_c			
✓ x			
☐ b_c			
☐ b_r			

```
for n in nodes do
    sum x(_,_,n) = 4
end

for r in rows do
    for c in cols do
        sum x(r,c,_) <= 1
    end
end
```

Figure 2: Non-overlapping nodeboxes with size four.

Now hit Run and wait for a few seconds. When PIGRA is finished it will show you the second tab on the left. It should look something like the drawing in Figure 1, note that we unchecked the b and e variables.

While all nodes are modelled with boxes of size 4 we never forbid them to overlap. In any real case we definitely would want the boxes to not intersect. Again only a bit of PGL code is enough to accomplish that:

```
**for** $r$ **in** $rows$ **do**
    **for** $c$ **in** $cols$ **do**
        $x(r, c, \_) \leq 1$
    **end**
**end**
```

This time we iterate over the rows and columns and don't restrict the nodes. In the end we get for each grid point (r, c) that there is at most one node at this coordinate. Hit Run again and this time the output should looks like the one in Figure 2.

3 Visibility Representations

Coming soon.

4 PGL

In this section we want to give a full reference of the statements currently implemented in PIGRA. For a quick overview look into cheatsheet.pdf. In PGL you work on sets. There are five basic sets, derived from the grid and the given graph:

- nodes
- edges
- objects (nodes ∪ edges)
- rows
- cols

With these sets you can define your own variables, iterate over them or do calculations. The following table displays all of PGL's control and mathematical statements. Where we write (...) you can put in restrictions, which will be explained later. The last and perhaps most important

Define a variable	**def** <name>(...)

For loops	**for** <name> **in** <set or variable> **do**
	....
	end

Sums	**sum** variable(...)
Assigns	mathexpr <= | >= | = mathexpr
Mathexpr	$number * (sum \mid number \mid variable(...) + -sum \mid number \mid variable(...))$

building block of PGL are restrictions. A restriction basically allows you to pick elements from a set and work only on them. For example if you want to iterate over all edges with red color or all the neighbours of a node. There are mainly three different types of restrictions. First of all you can restrict integer sets by giving an interval [1..5], which would be all numbers from 1 to 5. So if you wanted to define a variable on the first five rows and last five columns you could do it like this:

def var(rows[1 .. 5], cols[$\#cols - 6$ .. $\#cols - 1$)

If you later want to restrict the just defined variable *var* again, lets say on the second to fourth column you can do it like this:

def var([2 .. 4], _)

Here the *rows* specifier is omitted, since it is clear that the first set *var* is specified on are *rows*. The _ states that we want to use all the elements of the set, the variable is defined on at this position. *attribute-restrictions* are the next type of restrictions we will see. They are used to restrict a set of graph elements like *edges* through an attribute like color or label:

<set|name> **with|without** <*attribute*> value

As you can see we can either include or exclude an element based on the value of an attribute. The currently available attributes are:

- color <red,green,blue,black,white,cyan,purple,yellow,orange>
- label <string>
- nodes <name>

- edges <name>

The last two might need an explanation. They are only usable with *without* and allow you to exclude a specific set of nodes or edges from another set of node or edges. For example if you iterate over all edges and in each step you want to do something with all nodes, except the ones in (u, v), then you can exclude the nodes of this edge:

```
def x(nodes)
for e in edges do
    x(nodes without nodes e) = 1
end
```

What we call *property-restrictions* is the last type of restrictions. Again it works on graph elements, but this time we restrict properties of nodes or edges like "Pick all neighbours of this node *n*" or "Choose all incident edges of this node". The genereal syntax would be:

<property> of <set|name>

Currently PGL supports the following properties:

- neighbours (of node)
- incident edges (of node)
- source (of edge)
- target (of edge)

Of course you can use attribute- and property-restrictions together. In general if you are not giving braces PGL will evaluate all property-restrictions before evaluating any attribute-restrictions.

Finally you can use "mathematical" functions on sets, but there are only *max* and *min* so far, which will give you the maximum or the minimum value of a row or column set.